Water potablity Prediction

2021/8/24

Jhan-Yi LIN

Preview of the dataset and target

The target of this project is to predict whether a water body is potable or not, based on it's water quality.

These are the qualities that will be used in the prediction :

- ph value : The acidic or alkaline condition of water status
- Hardness : Capacity of water to precipitate soap in mg/L
- Solids : Total dissolved solids in ppm
- Chioramines : Amount of Chloramines in ppm
- Sulfate : Amount of Sulfates dissolved in mg/L
- Conductivity : Electrical conductivity of water in µS/cm
- Organic Carbon : Amount of organic carbon in ppm
- Trihalomethanes : Amount of Trihalomethanes in µg/L
- Turbidity : Measure of light emiting property of water in NTU (Nephelometric Turbidity Units)
- Potability : Indicates if water is safe for human consumption

Data Cleaning

In the first section, some basic data cleaning will be conducted to handle the missing value in the dataset, and the skewness of each attributes.

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import StandardScaler
```

In [3]:
```
df = pd.read_csv('water_potability.csv')
df
```

Out[3]:

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	T

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	T
0	NaN	204.890455	20791.318981	7.300212	368.516441	564.308654	10.379783	
1	3.716080	129.422921	18630.057858	6.635246	NaN	592.885359	15.180013	
2	8.099124	224.236259	19909.541732	9.275884	NaN	418.606213	16.868637	
3	8.316766	214.373394	22018.417441	8.059332	356.886136	363.266516	18.436524	
4	9.092223	181.101509	17978.986339	6.546600	310.135738	398.410813	11.558279	
...	...	...	...	...	...	...	...	
3271	4.668102	193.681735	47580.991603	7.166639	359.948574	526.424171	13.894419	
3272	7.808856	193.553212	17329.802160	8.061362	NaN	392.449580	19.903225	
3273	9.419510	175.762646	33155.578218	7.350233	NaN	432.044783	11.039070	
3274	5.126763	230.603758	11983.869376	6.303357	NaN	402.883113	11.168946	
3275	7.874671	195.102299	17404.177061	7.509306	NaN	327.459760	16.140368	

3276 rows × 10 columns

In [4]:
```
df.isnull().sum()
```

Out[4]:
```
ph                491
Hardness            0
Solids              0
Chloramines         0
Sulfate           781
Conductivity        0
Organic_carbon      0
Trihalomethanes   162
Turbidity           0
Potability          0
dtype: int64
```

In [5]:
```
df.describe()
```

Out[5]:

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_car
count	2785.000000	3276.000000	3276.000000	3276.000000	2495.000000	3276.000000	3276.00(
mean	7.080795	196.369496	22014.092526	7.122277	333.775777	426.205111	14.28
std	1.594320	32.879761	8768.570828	1.583085	41.416840	80.824064	3.30
min	0.000000	47.432000	320.942611	0.352000	129.000000	181.483754	2.20(
25%	6.093092	176.850538	15666.690297	6.127421	307.699498	365.734414	12.06
50%	7.036752	196.967627	20927.833607	7.130299	333.073546	421.884968	14.21
75%	8.062066	216.667456	27332.762127	8.114887	359.950170	481.792304	16.557
max	14.000000	323.124000	61227.196008	13.127000	481.030642	753.342620	28.30(

As we can see there are some missing value in the ph, Sulfate and Trihalomethanes. The missing value of sulfate is about 24% of the total population, assign a value to them might cause a bias,

and since the standard deviation of ph and Trihalomethanes are larger than their interquartile, I will drop the null value they contains rather than fill them with their mean.

In [6]:
```
df1 = df.dropna(axis=0).reset_index(drop=True)
df1
```

Out[6]:

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon
0	8.316766	214.373394	22018.417441	8.059332	356.886136	363.266516	18.436524
1	9.092223	181.101509	17978.986339	6.546600	310.135738	398.410813	11.558279
2	5.584087	188.313324	28748.687739	7.544869	326.678363	280.467916	8.399735
3	10.223862	248.071735	28749.716544	7.513408	393.663396	283.651634	13.789695
4	8.635849	203.361523	13672.091764	4.563009	303.309771	474.607645	12.363817
...	...	...	...	...	...	...	...
2006	8.989900	215.047358	15921.412018	6.297312	312.931022	390.410231	9.899115
2007	6.702547	207.321086	17246.920347	7.708117	304.510230	329.266002	16.217303
2008	11.491011	94.812545	37188.826022	9.263166	258.930600	439.893618	16.172755
2009	6.069616	186.659040	26138.780191	7.747547	345.700257	415.886955	12.067620
2010	4.668102	193.681735	47580.991603	7.166639	359.948574	526.424171	13.894419

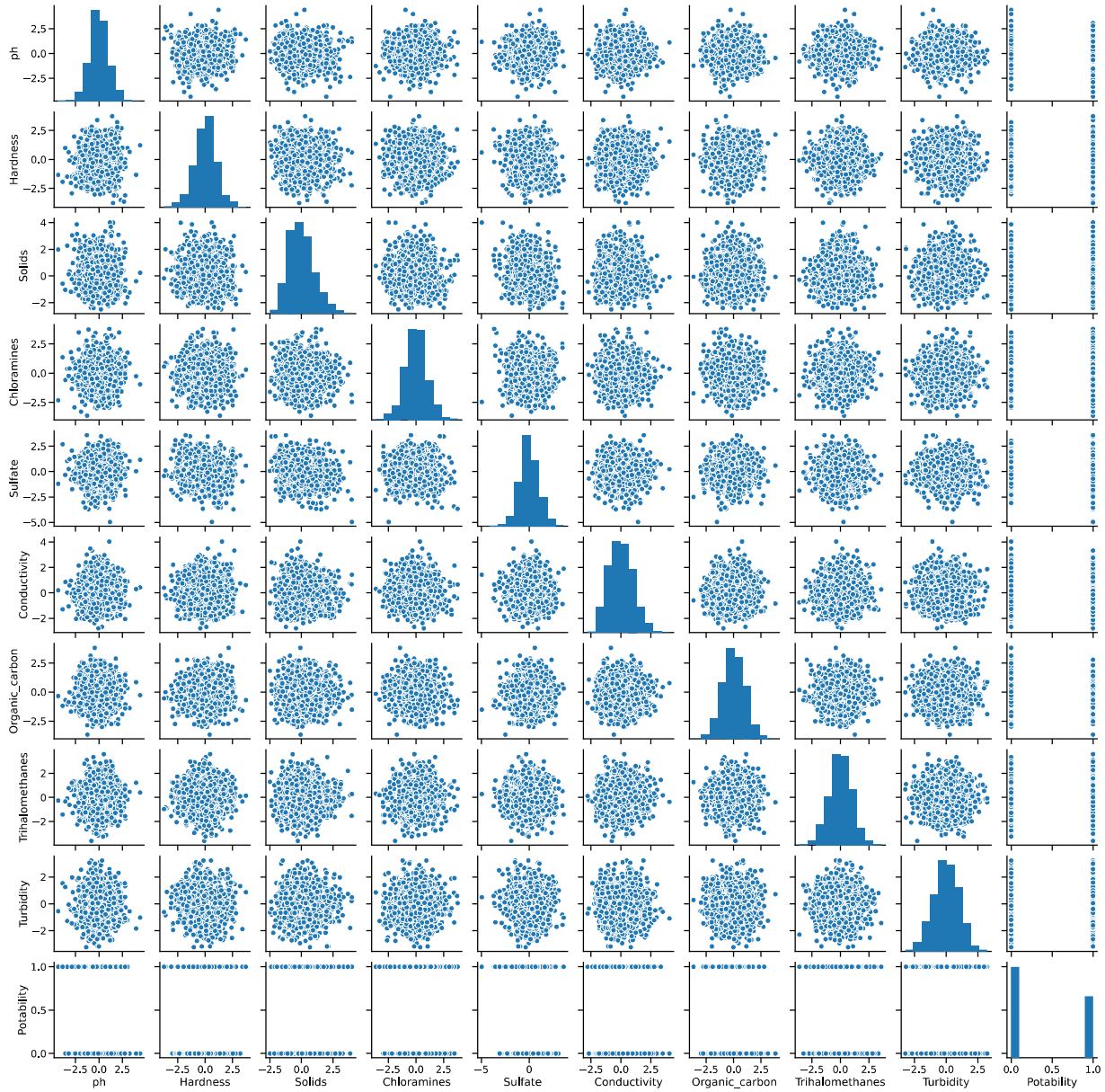
2011 rows × 10 columns

There are 2011 samples remain in the dataset after droping all the missing values, which is still sufficient to work on. Now let's plot out the distribution of the dataset to check the skewness. I set my skew limit to 0.75, this means every attributes with a skewness exceed the limit need to be modified to prevent a bias issue.

In [361…
```
print(df1.skew().sort_values())
sns.pairplot(df1)
```

```
Hardness          -0.085237
Trihalomethanes   -0.051422
Sulfate           -0.046558
Turbidity         -0.033051
Organic_carbon    -0.020018
Chloramines        0.012976
ph                 0.048947
Conductivity       0.266869
Potability         0.394614
Solids             0.595894
dtype: float64
```

Out[361…
```
<seaborn.axisgrid.PairGrid at 0x1c1c45df1c0>
```



As we can see in the plot, the distribution of "Solids" is a bit positive skewed. But since the skewness dosen't exceed the limit, it's fair to believe it won't cause a very big problem in prediction. So now I'll standardrize it, and then start to work on the classifying process.

In [8]:
```
df1_s = pd.DataFrame(StandardScaler().fit_transform(df1))
df1.iloc[:, :-1] = df1_s.iloc[:, :-1]
df1
```

Out[8]:

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	Trihalc
0	0.782466	0.564114	0.011687	0.583804	0.574378	-0.783962	1.227032	
1	1.275463	-0.455653	-0.455835	-0.370947	-0.560480	-0.348429	-0.842154	
2	-0.954835	-0.234614	0.790645	0.259104	-0.158911	-1.810063	-1.792340	
3	1.994902	1.596951	0.790764	0.239248	1.467140	-1.770608	-0.170876	
4	0.985323	0.226606	-0.954313	-1.622878	-0.726179	0.595858	-0.599824	
...	...	...	...	...	...	...	...	
2006	1.210411	0.584770	-0.693978	-0.528284	-0.492625	-0.447578	-1.341281	
2007	-0.243774	0.347964	-0.540564	0.362137	-0.697038	-1.205321	0.559422	

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	Trihalo
2008	2.800492	-3.100365	1.767503	1.343596	-1.803476	0.165656	0.546021	
2009	-0.646160	-0.285317	0.488576	0.387023	0.302843	-0.131852	-0.688929	
2010	-1.537172	-0.070075	2.970287	0.020386	0.648718	1.238006	-0.139372	

2011 rows × 10 columns

Classification

The first step to classification is to split the train and test data. If we take a look of the distribution of the potability, we can see the ratio of unpotable and potable samples is about 3:2, so I'll use the stratified shuffle split to ensure the train and test data both remain the same ratio.

In [9]:
```
df1.Potability.value_counts()

x = df1.drop('Potability', axis=1)
y = df1.Potability
```

In [10]:
```
print(x.shape, y.shape)
```

```
(2011, 9) (2011,)
```

In [17]:
```
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.model_selection import StratifiedKFold
skf = StratifiedKFold(n_splits = 4, shuffle = True, random_state=712)

sss = StratifiedShuffleSplit(test_size=0.3, random_state=712)
train_idx, test_idx = next(sss.split(x, y))
x_train = x.loc[train_idx]
y_train = y.loc[train_idx]
x_test  = x.loc[test_idx]
y_test  = y.loc[test_idx]
```

In [163…
```
y_train.value_counts(normalize=True).sort_index()
```

Out[163…
```
0    0.597015
1    0.402985
Name: Potability, dtype: float64
```

In [167…
```
y_test.value_counts(normalize=True).sort_index()
```

Out[167…
```
0    0.596026
1    0.403974
Name: Potability, dtype: float64
```

Logistic regression

First, I'll fit the data with logistic regression.

In [352…
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
```

```
def measure_error(y_true, y_pred, label):
    return pd.Series({'accuracy':accuracy_score(y_true, y_pred),
                      'precision': precision_score(y_true, y_pred),
                      'recall': recall_score(y_true, y_pred),
                      'f1': f1_score(y_true, y_pred, average='binary')},
                      name=label)
```

In [353…

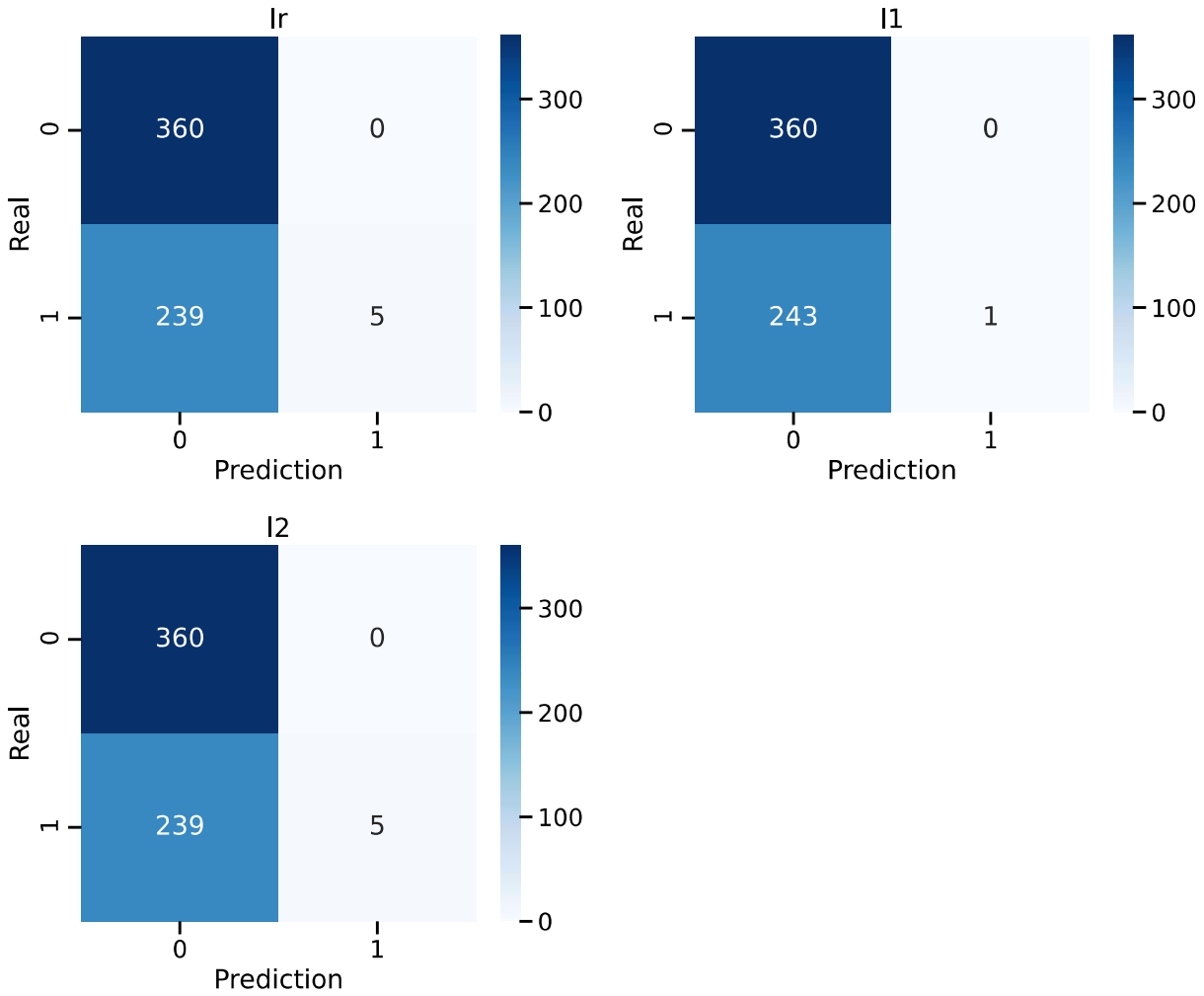
```
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import LogisticRegressionCV
from sklearn.model_selection import cross_val_predict

lr = LogisticRegression(solver='liblinear')
lr_l1 = LogisticRegressionCV(Cs=10, cv=4, penalty='l1', solver='liblinear')
lr_l2 = LogisticRegressionCV(Cs=10, cv=4, penalty='l2', solver='liblinear')

labels = ['lr', 'l1', 'l2']
models = [lr, lr_l1, lr_l2]
train_m = list()
test_m = list()
train_cm = dict()
test_cm = dict()

for lab, mod in zip(labels, models) :
    mod.fit(x_train, y_train)
    y_train_pred = mod.predict(x_train)
    y_test_pred = mod.predict(x_test)
    train_m.append(measure_error(y_train, y_train_pred, lab+'_train'))
    test_m.append(measure_error(y_test, y_test_pred, lab+'_test'))
    train_cm[lab] = confusion_matrix(y_train, y_train_pred)
    test_cm[lab] = confusion_matrix(y_test, y_test_pred)


train_m = pd.concat(train_m, axis=1)
test_m = pd.concat(test_m, axis=1)

train_m, test_m
```

Out[353…

```
(           lr_train  l1_train  l2_train
accuracy    0.601279  0.599147  0.601279
precision   1.000000  1.000000  1.000000
recall      0.010582  0.005291  0.010582
f1          0.020942  0.010526  0.020942,
             lr_test   l1_test   l2_test
accuracy    0.604305  0.597682  0.604305
precision   1.000000  1.000000  1.000000
recall      0.020492  0.004098  0.020492
f1          0.040161  0.008163  0.040161)
```

In [354…

```
fig, axList = plt.subplots(nrows=2, ncols=2)
axList = axList.flatten()
fig.set_size_inches(12, 10)

axList[-1].axis('off')

for ax,lab in zip(axList[:-1], labels):
    sns.heatmap(test_cm[lab], ax=ax, annot=True, fmt='d', cmap='Blues');
    ax.set(xlabel='Prediction', ylabel='Real', title=lab);

plt.tight_layout()
```



The model seems to be underfitted, since it tends to predict all the outcome to be class 0, which is unpotable.

If we take a deeper loo at the outcome of both train and test samples, we can see the accuracys are barely the same, and the precisions are all eqaul to 1. This prove the point that the model tends to just simply predict all the samples to be class 0.

SVM

Considering the outcome of logistic regression, the model might have a tendecy to predict all samples as class 0 due to the unbalance ratio of the two classes of sample. Hence, the class_weight of model will be set to "balanced" to handle this problem.

In [355…

```
from sklearn.svm import SVC
test_m = list()
test_cm = dict()
train_cm = dict()

gammas = [.01, .05, .1, .5, 1, 2]
for gamma in gammas:
    svm = SVC(class_weight='balanced', kernel='rbf', gamma=gamma, probability=True)
    svm.fit(x_train, y_train)
    y_test_pred = svm.predict(x_test)
    y_train_pred = svm.predict(x_train)
    test_m.append(measure_error(y_test, y_test_pred, gamma))
    test_cm[gamma] = confusion_matrix(y_test, y_test_pred)
    train_cm[gamma] = confusion_matrix(y_train, y_train_pred)
```

```
test_m = pd.concat(test_m, axis=1)
test_m
```

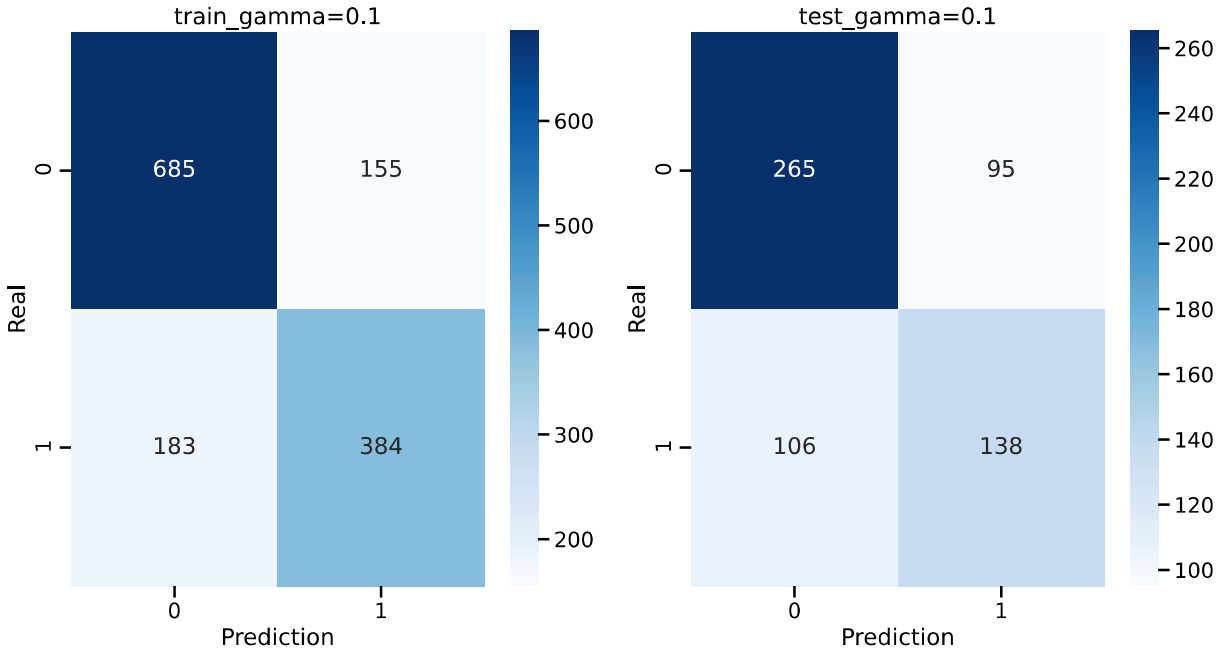
Out[355…

	0.01	0.05	0.10	0.50	1.00	2.00
accuracy	0.645695	0.653974	0.667219	0.627483	0.591060	0.596026
precision	0.678571	0.587940	0.592275	0.544601	0.484536	0.500000
recall	0.233607	0.479508	0.565574	0.475410	0.192623	0.024590
f1	0.347561	0.528217	0.578616	0.507659	0.275660	0.046875

In [356…

```
best_svm = SVC(class_weight='balanced', kernel='rbf', gamma=0.1, probability=True)
best_svm.fit(x_train, y_train)
y_train_pred = best_svm.predict(x_train)
y_test_pred = best_svm.predict(x_test)
print(measure_error(y_train_pred, y_train, 'train_gamma=0.1'), measure_error(y_test_


fig, axList = plt.subplots(nrows=1, ncols=2)
fig.set_size_inches(16, 8)
ax1 = sns.heatmap(train_cm[.1], ax=axList[0], annot=True, fmt='d', cmap='Blues');
ax1.set(xlabel='Prediction', ylabel='Real', title='train_gamma=0.1');
ax2 = sns.heatmap(test_cm[.1], ax=axList[1], annot=True, fmt='d', cmap='Blues');
ax2.set(xlabel='Prediction', ylabel='Real', title='test_gamma=0.1');
```

```
accuracy     0.759773
precision    0.677249
recall       0.712430
f1           0.694394
Name: train_gamma=0.1, dtype: float64 accuracy     0.667219
precision    0.565574
recall       0.592275
f1           0.578616
Name: test_gamma=0.1, dtype: float64
```



The best outcome of SVM is with the gamma as 0.1. As we can see in the confusion matrix, is already way better than the logistic regression.

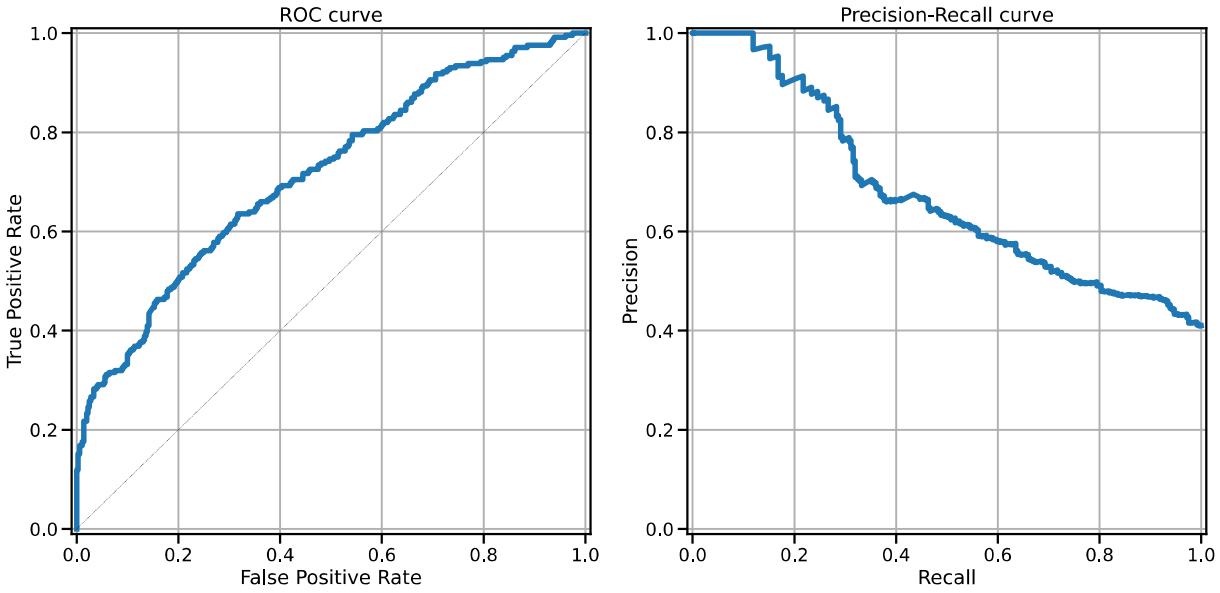
With a check of the confusion matrix of the train data, it seems like the model doesn't have a issue with overfitting.

Let's also takke a look at the ROC and Precision-Recall curves.

In [357…

```
from sklearn.metrics import roc_curve, precision_recall_curve, roc_auc_score
sns.set_context('talk')

fig, axList = plt.subplots(ncols=2)
fig.set_size_inches(16, 8)

# Get the probabilities for each of the two categories
y_prob = best_svm.predict_proba(x_test)

# Plot the ROC-AUC curve
ax = axList[0]

fpr, tpr, thresholds = roc_curve(y_test, y_prob[:,1])
ax.plot(fpr, tpr, linewidth=5)
# It is customary to draw a diagonal dotted line in ROC plots.
# This is to indicate completely random prediction. Deviation from this
# dotted line towards the upper left corner signifies the power of the model.
ax.plot([0, 1], [0, 1], ls='--', color='black', lw=.3)
ax.set(xlabel='False Positive Rate',
       ylabel='True Positive Rate',
       xlim=[-.01, 1.01], ylim=[-.01, 1.01],
       title='ROC curve')
ax.grid(True)

# Plot the precision-recall curve
ax = axList[1]

precision, recall, _ = precision_recall_curve(y_test, y_prob[:,1])
ax.plot(recall, precision, linewidth=5)
ax.set(xlabel='Recall', ylabel='Precision',
       xlim=[-.01, 1.01], ylim=[-.01, 1.01],
       title='Precision-Recall curve')
ax.grid(True)

plt.tight_layout()
print('auc :', roc_auc_score(y_test, y_test_pred))
```

```
auc : 0.6508424408014573
```



As we see, the ROC indicates the model has a much better explanation and predict ability than guessing randomly. Also, the AUC score is about 0.65, which is not bad, but maybe we can have a better performance with a more complex model.

Decision Tree

Let's try the Decision Tree since it tends to come out with a more complex model. Since decision tree is unstable and come with different out come every time, a random state will be set to make sure the result will be the same.

Noted that the scoring rule is using average precision. If we think about the target of the test, which is predicting the potability of water body. We would like the model to be more precisely when labeling a water body ass potable, since it's cause much more severe issue when we wrongly labeled a unpotable water body as potable, rather than labeled a potable water body as unpotable.

This means that we want a higher precision. But if we set the scoring rule to maximize the precision, the model would tend to just label all samples as unpotable unless it's very confident about a potable one.

Using the average precision can balance these issue.

In [358…
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
tree = DecisionTreeClassifier(class_weight='balanced', random_state=712)
tree.fit(x_train, y_train)
param_grid = {'max_depth':range(1, tree.tree_.max_depth+1, 2),
              'max_features': range(1, len(tree.feature_importances_)+1)}

gsdt = GridSearchCV(DecisionTreeClassifier(random_state=712),
                    param_grid=param_grid,
                    scoring='average_precision',
                    n_jobs=-1, cv=10)
gsdt.fit(x_train, y_train)
```

Out[358…
```
GridSearchCV(cv=10, estimator=DecisionTreeClassifier(random_state=712),
             n_jobs=-1,
             param_grid={'max_depth': range(1, 26, 2),
                         'max_features': range(1, 10)},
             scoring='average_precision')
```

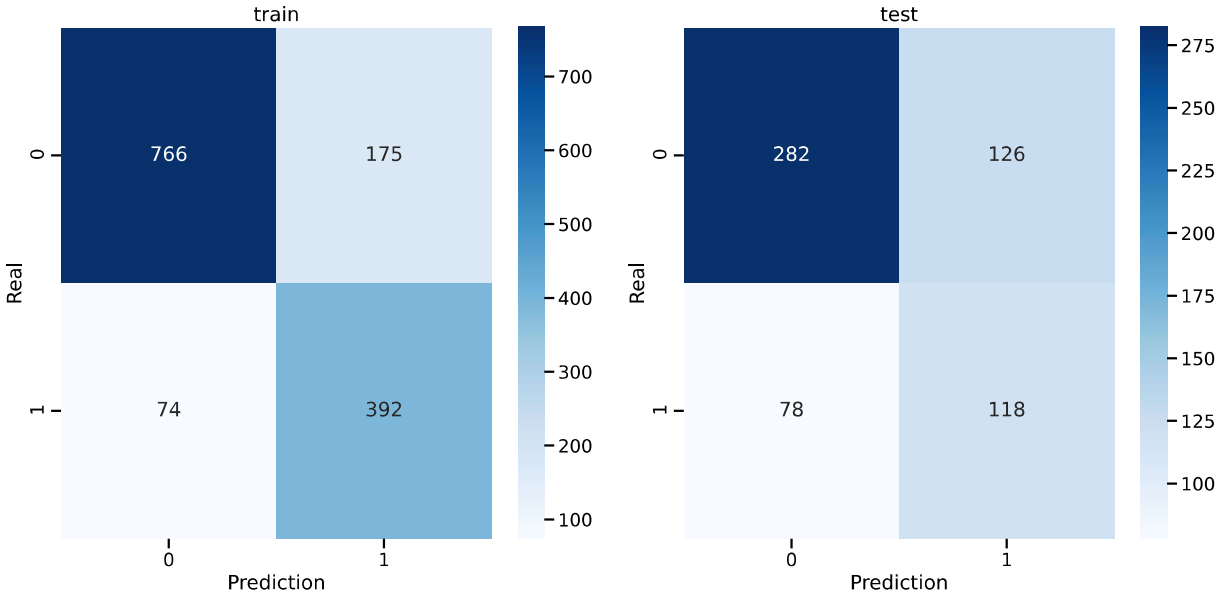
In [359…
```
print('max_features :', gsdt.best_estimator_.max_features,
      '\nmax_depth :', gsdt.best_estimator_.tree_.max_depth)
```

```
max_features : 9
max_depth : 11
```

In [360…
```
best_tree = DecisionTreeClassifier(max_features=9, max_depth=11,
                                   class_weight='balanced', random_state=712)
best_tree.fit(x_train, y_train)
y_train_pred = best_tree.predict(x_train)
y_test_pred = best_tree.predict(x_test)
print(measure_error(y_train_pred, y_train, 'train_max_depth=5'), measure_error(y_tes

cm = dict()
cm['train'] = confusion_matrix(y_train_pred, y_train)
cm['test'] = confusion_matrix(y_test_pred, y_test)

fig, axList = plt.subplots(nrows=1, ncols=2)
fig.set_size_inches(16, 8)

ax1 = sns.heatmap(cm['train'], ax=axList[0], annot=True, fmt='d', cmap='Blues');
ax1.set(xlabel='Prediction', ylabel='Real', title='train');
```

```
ax2 = sns.heatmap(cm['test'], ax=axList[1], annot=True, fmt='d', cmap='Blues');
ax2.set(xlabel='Prediction', ylabel='Real', title='test');

plt.tight_layout()
```

```
accuracy     0.823028
precision    0.691358
recall       0.841202
f1           0.758955
Name: train_max_depth=5, dtype: float64 accuracy     0.662252
precision    0.483607
recall       0.602041
f1           0.536364
Name: test_max_depth=5, dtype: float64
```



If we compare the confusion matrix of decision tree with the one using SVM, we can tell that based on the training data, decision tree does predict more precise. But if we look at the test data, SVM has a better performance.

This might indicates that in this case , a more complex model might overfit and doesn't perform better compares to SVM. So we would just use the SVM model a our best estimator.

Further Steps

Learn about the standard of which water quality indicator will have a larger effect in determining potability, then conduct a classification with pre-weighted attributes might have a better results.

This concludes the project.